Tutorial 4

Working with Objects & Arrays

What are Objects?

- JavaScript's fundamental datatype is the object .
- Objects are things we deal with everyday. Javascript deals with objects, as do most programming languages, and these languages are called object oriented or for short, OOP.
- Objects are just a way of representing data.
- Objects are composite types. They provide a way to organize a collection of data into a single unit. Object oriented languages, such as C++ and Java, bundle up data into a variable and call it an object. So does JavaScript. When you learn about objects, they are usually compared to real world things, like a cat, a book, a triangle. Using the English language to describe an object, the object itself would be like a noun.
- JavaScript objects are dynamic—properties can usually be added and deleted but they can be used to simulate the static objects and "structs" of statically typed languages.
- Objects are mutable and are manipulated by reference rather than the by value.
- JavaScript supports several types of objects. They are:
  - **1. User-defined objects defined by the programmer.**
  - **2. Core or built-in objects, such as Date, String, and Number**
  - **3. Browser and Document objects.**

Creating Object Types

- **Object Instances are ideally suited to storing and transmitting data around an application.**

- **The first is to use the new operator.**
  - **var person = new Object();// A function used in this way is called a constructorand serves to initialize a newly created object.**
  - **person.name = "Vivek";**
  - **person.age = 99;**

- **The other way is to use object literal notation.**
  - **var person = {**
  - **name : "Vivek",**
  - **age : 99**
  - **};**

- **Though it's acceptable to use either method of creating Object instances, developers tend to favor object literal notation, because it requires less code and visually encapsulates all related data.**

Elements, Properties, Methods

- **Instances of Object aren't very useful on their own, but the concepts are important to understand, because, similar to java.lang.Object in Java, the Object type in ECMAScript is the base from which all other objects are derived. All of the properties and methods of the Object type are also present on other, more specific objects like Arrays, Date, RegularExpressions etc.**

- **Objects contain properties.**

- **A property of an object can contain a function, because functions are just data.**

```
var dog = {
        first name: 'Nicholas',

        talk: function(){  // In this case we would call the function as method.
                alert('Woof, woof!');
                }
};
```

Accessing Object Properties

- Typically accessed using dot notation, which is common to many object-oriented languages.
- it's also possible to access properties via bracket notation.
  - alert(person["name"]); //"Nicholas"
  - alert(person.name); //"Nicholas"
- You can also use bracket notation when the property name contains characters that would be either a syntax error or a keyword/reserved word. For example:
- person["first name"] = "Nicholas";//Since the name "first name" contains a space, you can't use dot notation to access it.
- Generally speaking, dot notation is preferred unless variables are necessary to access properties by name.

Calling an Object's Methods.

- **Because a method is just a property that happens to be a function, you can access methods in the same way as you would access properties: using the dot notation or using square brackets. Calling (invoking) a method is the same as calling any other function: just add parentheses after the method name, which effectively say "Execute!".**

```
var hero = {
        breed: 'Turtle',
        occupation: 'Ninja',
        say: function() {
                return 'I am ' + hero.occupation;
                }
        }
>>> hero.say();
```

Object Models and the Dot Syntax

- An object model is a hierarchical tree-like structure used to describe all of the components of an object.
- When accessing an object in the tree, the object at the top of the tree is the root or parent of all parents. If there is an object below the parent, it is called the child and if the object is on the same level, it is a sibling. A child can also have children, etc. etc.
- A dot is used to separate the objects when descending the tree; e.g. a parent is separated from its child with a dot. In the following example, the pet object is subdivided into subordinate or child objects: a cat and a dog. The cat and the dog objects each have properties associated with them. In order to navigate down the tree to the cat's name, for example, you would say pet.cat.name and to get the dog's breed you would say pet.dog.breed.

Objects Continued…

- Each Object instance has the following properties and methods:
- constructor — The function that was used to create the object. In the previous example, the constructor is the Object() function.
- hasOwnProperty(propertyName) — Indicates if the given property exists on the object instance (not on the prototype). The property name must be specified as a string (for example, o.hasOwnProperty("name")).
- isPrototypeOf(object) — Determines if the object is a prototype of another object.
- propertyIsEnumerable(propertyName) — Indicates if the given property can be enumerated using the for-in statement (discussed later in this chapter). As with
- hasOwnProperty(), the property name must be a string.
- toLocaleString() — Returns a string representation of the object that is appropriate for the locale of execution environment.
- toString() — Returns a string representation of the object.
- valueOf() — Returns a string, number, or Boolean equivalent of the object. It often returns the same value as toString().

Altering Properties/Methods

- **JavaScript is a dynamic language; it allows you to alter properties and methods of existing objects at any time. This includes adding new properties or deleting them. You can start with a blank object and add properties later.**

An empty object:>>> var hero = {};

Accessing a non-existing property: >>> typeof hero.breed >>"undefined"

Adding some properties and a method:

>>> hero.breed = 'turtle';

>>> hero.name = 'Leonardo';

>>> hero.sayName = function() {return hero.name;};

>>> Calling the method: >>> hero.sayName();"Leonardo"

Using this Value

- In the previous example, the method sayName() used hero.name to access the name property of the hero object. When you're inside a method though, there is another way to access the object this method belongs to: by using the special value this.

```
var hero = {
        name: 'Rafaelo',
        sayName: function() {
                return this.name;
                }
        }
>>> hero.sayName();
"Rafaelo"
```

- So when you say this, you are actually saying "this object" or "the current object".

The Object Oriented Paradigm

Object Oriented Programming

- **Objects interacting with one another through the methods and properties**
- **Used to store data, structure applications into modules and keeping code clean**

```
var jane = {
    name: 'John',
    yearOfBirth: 1990,
    isMarried : true

};
```

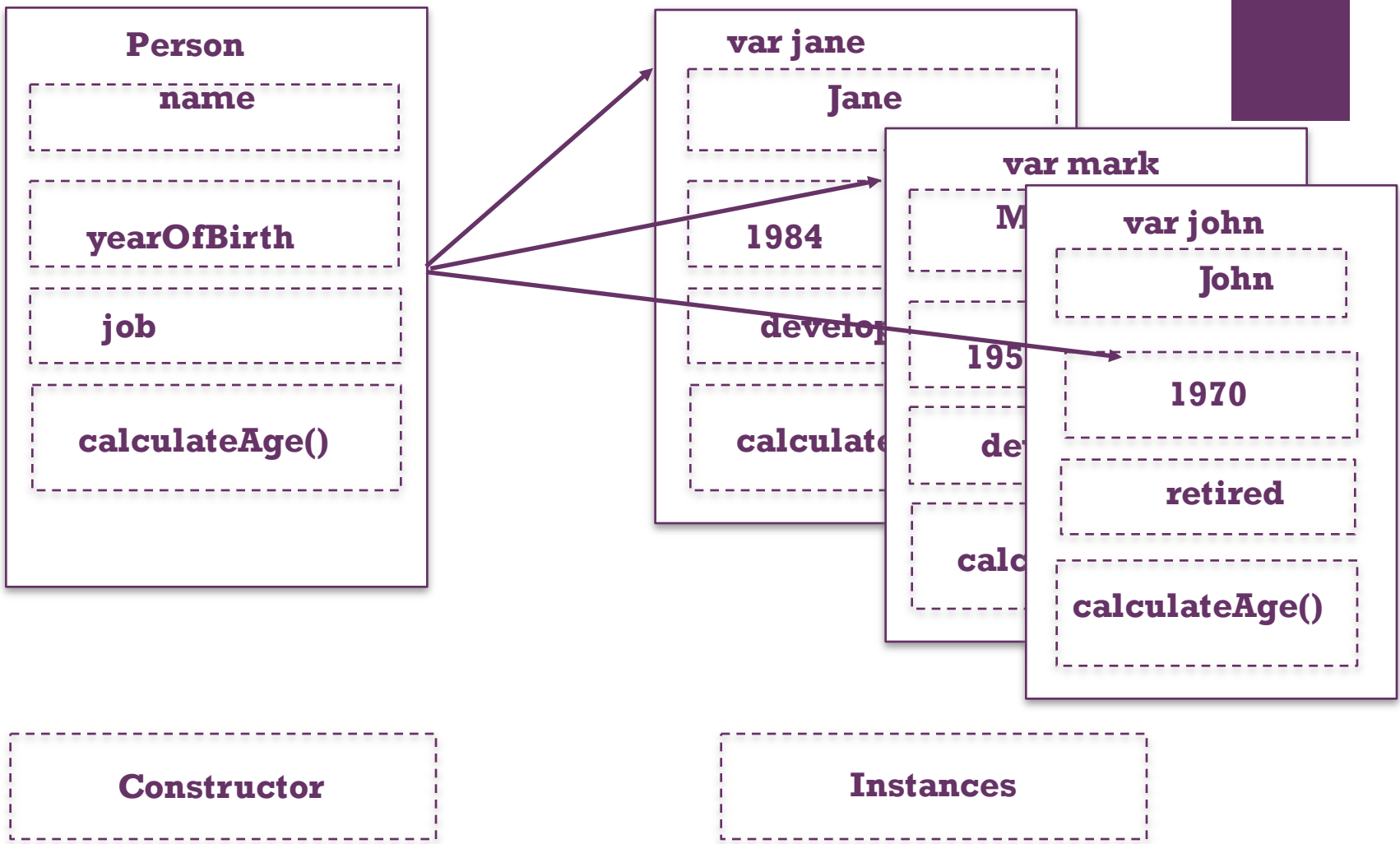
```
var vik = {
    name: 'Vik',
    yearOfBirth: 1980,
    isMarried : true

};
```

```
var doe = {
    name: 'Doe',
    yearOfBirth: 1976,
    isMarried : false

};
```

Constructors and Instances



The Constructor Functions.

- The only difference between constructor functions and other functions is the way in which they are called.
- Constructors are, after all, just functions.
- There's no special syntax to define a constructor that automatically makes it behave as such.
- Any function that is called with the new operator acts as a constructor, whereas any function called without it acts just as you would expect a normal function call to act.
- When an object is created, a special property is assigned to it behind the scenes—the constructor property. It contains a reference to the constructor function used to create this object.

constructor Property

- If you're familiar with object-oriented programming, you're likely accustomed to thinking of functions, methods, and class constructors as three separate things. In JavaScript, these are just three different usage patterns of one single construct: functions.
- The simplest usage pattern is the function call.
- Methods in JavaScript are nothing more than object properties that happen to be functions.
- The third use of functions is as constructors. Just like methods ad plain functions, constructors are defined with function.

The Global Object

- Previously we discussed global variables (and how you should avoid them) and also the fact that JavaScript programs run inside a host environment (the browser for example).
- Now that you know about objects, it is time for the whole truth: the host environment provides a global object and all global variables are actually properties of the global object.
- If your host environment is the web browser, the global object is called window.

Instance Of Property and Function that Return Objects

- **Instance Of Property : Using the instanceof operator, you can test if an object was created with a specific constructor function.**
- **Functions that Return Objects :**
- **In addition to using constructor functions and the new operator to create objects, you can also use a normal function and create objects without new. You can have a function that does some preparatory work and has an object as a return value.**

```
function factory(name) {
        return {
                name: name
        };
}
var o = factory('one');
o.name  // “One”
```

Passing Objects

- **When you copy an object or pass it to a function, you only pass a reference to that object. Consequently, if you make a change to the reference, you are actually modifying the original object.**

```
var original = {howmany:1};
var copy = original;
copy.howmany
copy.howmany = 100;
original.howmany
```

The same thing applies when passing objects to functions:

```
var original = {howmany:100};
var nullify = function(o){o.howmany = 0;}
nullify(original);
original.howmany
```

Comparing Objects

- When you compare objects, you'll get true only if you compare two references to the same object. Comparing two distinct objects that happen to have the exact same methods and properties will return false.
- var fido = {breed: 'dog'};
- var benji = {breed: 'dog'};
- Comparing them will return false.
- You can create a new variable mydog and assign one of the objects to it, this way mydog actually points to the same object.
- var mydog = benji;

Prototype

- The functions in JavaScript are objects and they contain methods and properties. Some of the methods that you are already familiar with are apply() and call() and some of the properties are length and constructor. Another property of the function objects is prototype.

- If you define a simple function foo() you can access its properties as you would do with any other object:
  - function foo(a, b){return a * b;}
  - foo.length //2
  - foo.constructor // Function()

- prototype is a property that gets created as soon as you define the function. Its initial value is an empty object.

- typeof foo.prototype // Object

- You can augment this empty object with properties and methods. They won't have any effect of the foo() function itself; they'll only be used when you use foo() as a constructor.

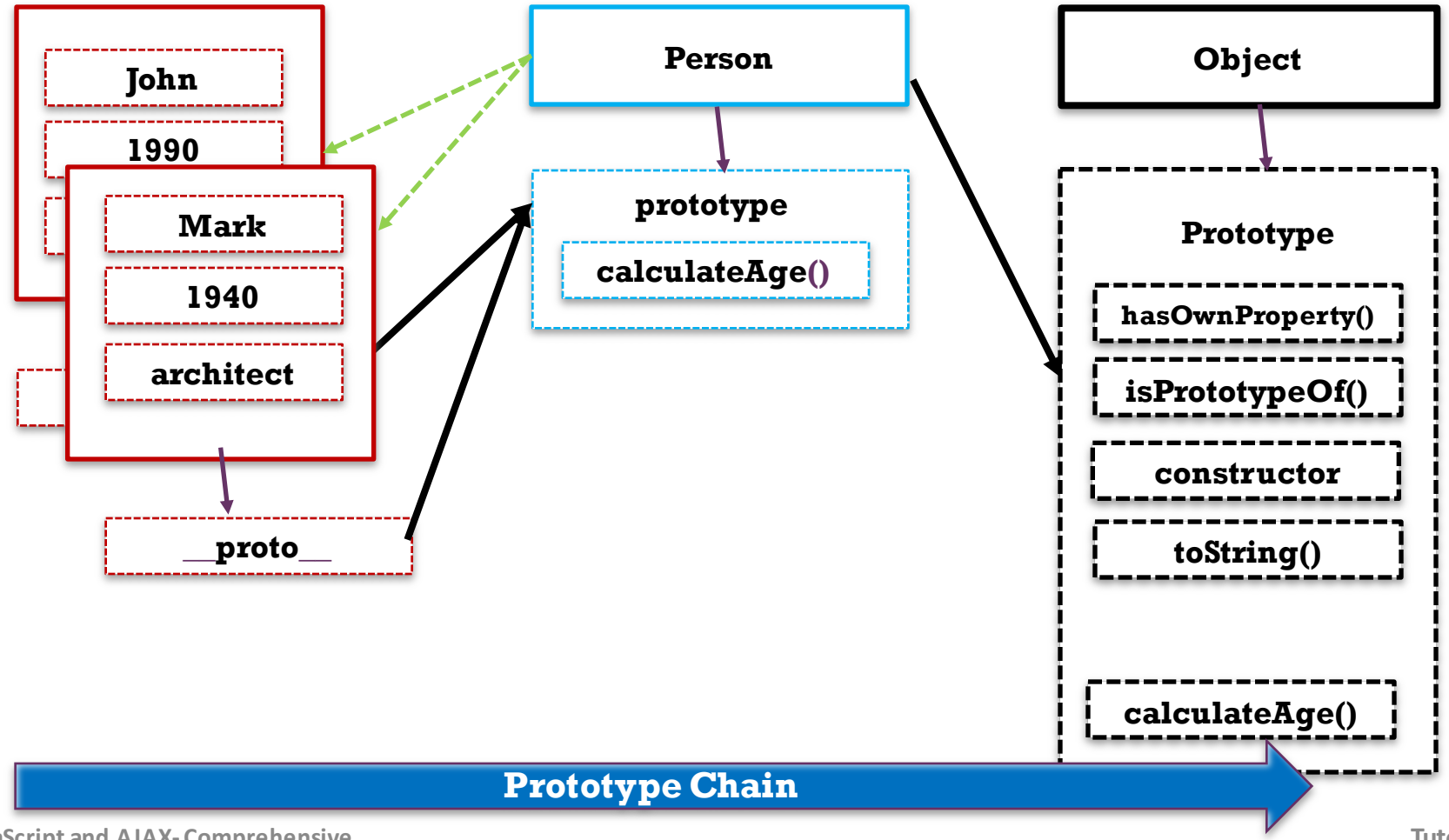
Adding Methods and Properties Using the Prototype

- We can define constructor functions which can be used to create (construct) new objects. The main idea was that inside a function invoked with new you have access to the value this, which contains the object to be returned by the constructor. Augmenting (adding methods and properties to) this object is the way to add functionality to the object being created.
- Adding methods and properties to the prototype property of the constructor function is another way to add functionality to the object being created.
- All the methods and properties you have added to the prototype are directly available as soon as you create a new object using the constructor
- It's important to note that the prototype is "live". Objects are passed by reference in JavaScript, and therefore the prototype is not copied with every new object instance. What does this mean in practice? It means that you can modify the prototype at any time and all objects (even those created before the modification) will inherit the changes.

So what is a prototype Really

Reference Points for other Objects

Prototypes and Prototype Chains



Summary

- Every JS Object has a property __proto__ property which makes inheritance possible in JS
- The prototype property of an object is where we add the properties and methods we want the instances to share.
- The Constructor's prototype property is **NOT** the __proto__ property of Constructor but its equal to all the __proto__ properties of the ALL the instances created through it.
- When a certain method or property is called, the search starts in the object itself and if it cannot be found, the search moves on to the __proto__ property, it its not found over there, continues in the __proto__ property of the parent. This continues until the method is found : **prototype chain.**

Own Properties versus prototype Properties

- In the example above getInfo() used this internally to address the object. It could've also used Gadget.prototype to achieve the same result:

- Gadget.prototype.getInfo = function() {

```
return 'Rating: ' + Gadget.prototype.rating + ', price: ' + Gadget.prototype.price;
};
```

- What's is the difference? To answer this question, let's examine how the prototype works in more detail.

- Let's again take our newtoy object:

- var newtoy = new Gadget('webcam', 'black');

- When you try to access a property of newtoy, say newtoy.name the JavaScript engine will look through all of the properties of the object searching for one called name and, if it finds it, will return its value. [newtoy.name // "webcam"].

- What if you try to access the rating property? The JavaScript engine will examine all of the properties of newtoy and will not find the one called rating. Then the script engine will identify the prototype of the constructor function used to create this object (same as if you do newtoy.constructor.prototype). If the property is found in the prototype, this property is used. [newtoy.rating //3]

Own Properties versus prototype Properties…..

- This would be the same as if you accessed the prototype directly. Every object has a constructor property, which is a reference to the function that created the object, so in our case:
  - >>> newtoy.constructor
- Gadget(name, color)
  - >>> newtoy.constructor.prototype.rating
- Now let's take this lookup one step further. Every object has a constructor. The prototype is an object, so it must have a constructor too. Which in turn has a prototype. In other words you can do:
  - >>> newtoy.constructor.prototype.constructor
- Gadget(name, color)
  - >>> newtoy.constructor.prototype.constructor.prototype
  - Object price=100 rating=3
- This might go on for a while, depending on how long the prototype chain is, but you eventually end up with the built-in Object() object, which is the highest-level parent. In practice, this means that if you try newtoy.toString() and newtoy doesn't have an own toString() method and its prototype doesn't either, in the end you'll get the Object's toString()
  - >>> newtoy.toString()
  - "[object Object]"

Overwriting Prototype's Property with Own Property

- As the above discussion demonstrates, if one of your objects doesn't have a certain property of its own, it can use one (if exists) somewhere up the prototype chain. What if the object does have its own property and the prototype also has one with the same name? The own property takes precedence over the prototype's.

- Let's have a scenario where a property name exists both as an own property and as a property of the prototype object:

```
function Gadget(name) {
        this.name = name;
}
```

- Gadget.prototype.name = 'foo';
  - "foo"

- Creating a new object and accessing its name property gives you the object's own name property.

```
>>> var toy = new Gadget('camera');
>>> toy.name;
"camera"
```

- If you delete this property, the prototype's property with the same name "shines through":
  - >>> delete toy.name;
  - true
  - >>> toy.name; //"foo"

Enumerable Properties

- If you want to list all properties of an object, you can use a for-in loop.
- Not all properties show up in a for-in loop. For example, the length (for arrays) and constructor properties will not show up. The properties that do show up are called enumerable. You can check which ones are enumerable with the help of the propertyIsEnumerable() method that every object provides.
- Prototypes that come through the prototype chain will also show up, provided they are enumerable. You can check if a property is an own property versus prototype's using the hasOwnProperty() method.
- propertyIsEnumerable() will return false for all of the prototype's properties, even those that are enumerable and will show up in the for-in loop.

isPrototypeOf()

- Every object also gets the isPrototypeOf() method. This method tells you whether that specific object is used as a prototype of another object.
  - Let's take a simple object monkey.

```
var monkey = {
        hair: true,
        feeds: 'bananas',
        breathes: 'air'
};
```

Now let's create a Human() constructor function and set its prototype property to point to monkey.

```
function Human(name) {
        this.name = name;
}
Human.prototype = monkey;
```

- Now if you create a new Human object called george and ask: "Is monkey george's prototype?", you'll get true.
  - >>> var george = new Human('George');
  - >>> monkey.isPrototypeOf(george)

Augmenting Built-in Objects

- The built-in objects such as the constructor functions Array, String, and even Object, and Function can be augmented through their prototypes, which means that you can, for example, add new methods to the Array prototype and in this way make them available to all arrays. Let's do this.

- Augmenting built-in objects through the prototype is a very powerful technique and you can use it to shape JavaScript any way you like. Because of its power, you should always thoroughly consider your options before using this approach.

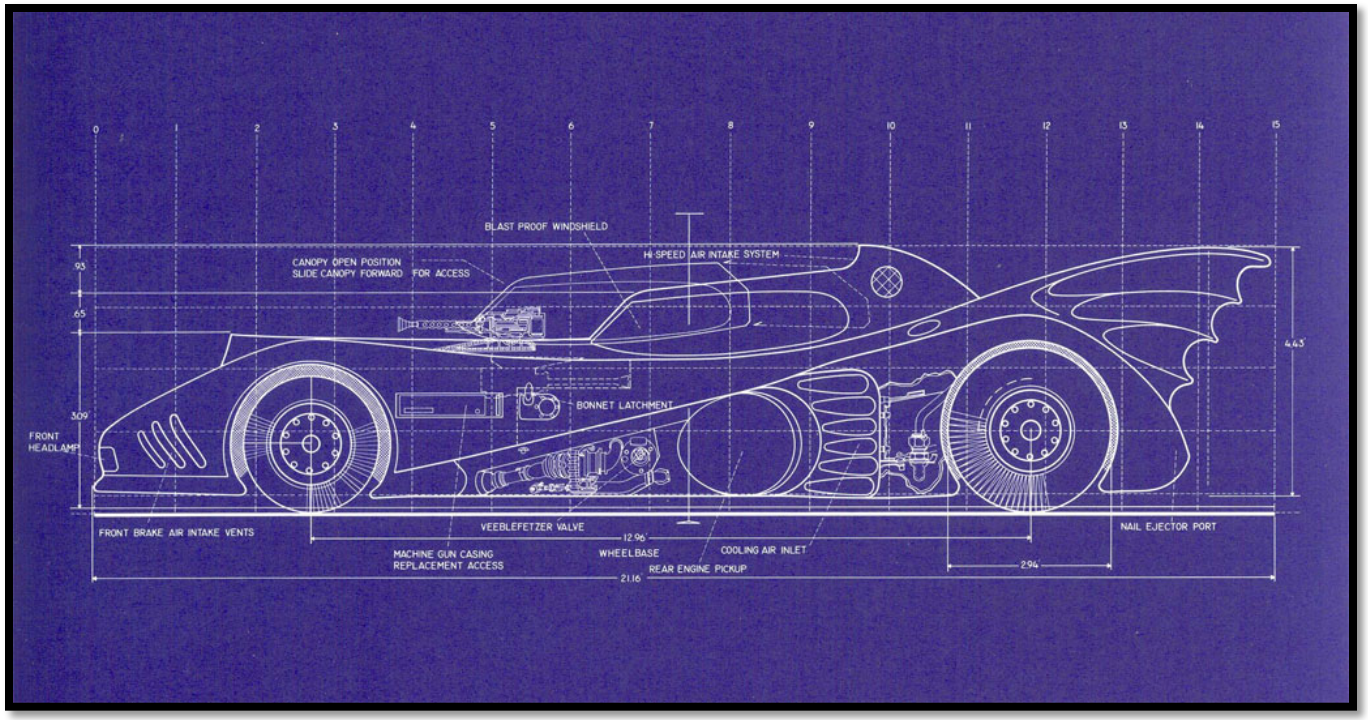
- Take the popular JavaScript library called Prototype. Its creator liked this approach so much that he even named the library after it. Using this library, you can work with JavaScript using methods very similar to the Ruby language.

- YUI (Yahoo! User Interface) library is another popular JavaScript library. Its creators are on the exact opposite side of the spectrum: they won't modify the built-in objects in any way. The reason is that once you know JavaScript, you're expecting it to work the same way, no matter which library you're using. Modifying core objects could only confuse the user of the library and create unexpected errors.

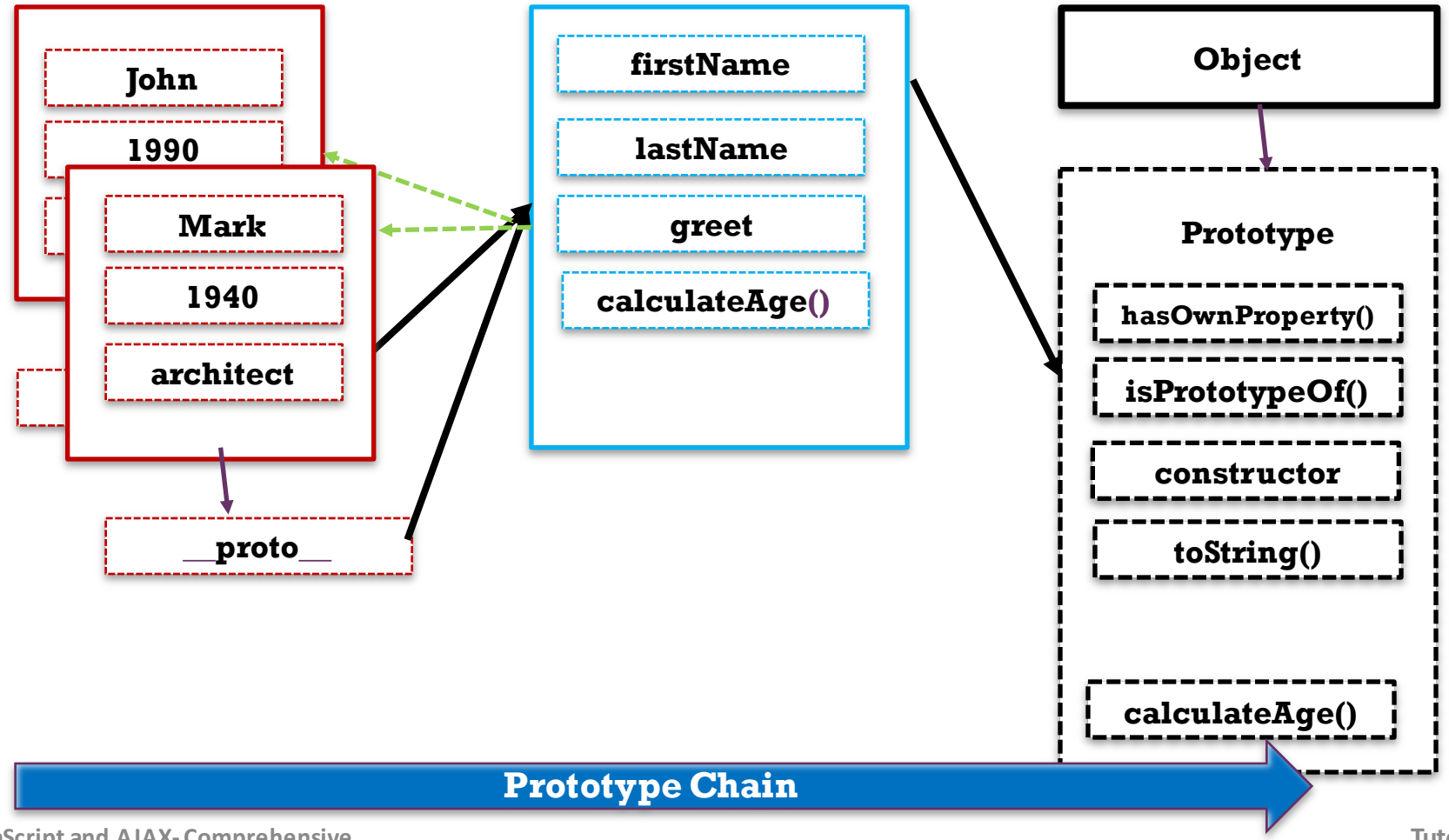
- The fact is that JavaScript changes and browsers come up with new versions that support more features. What you consider a missing feature today and decide to augment a prototype for, might be a built-in method tomorrow. In this case, your method is no longer needed. However, what if you have already written a lot of code that uses the method and your method is slightly different from the new built-in implementation?

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

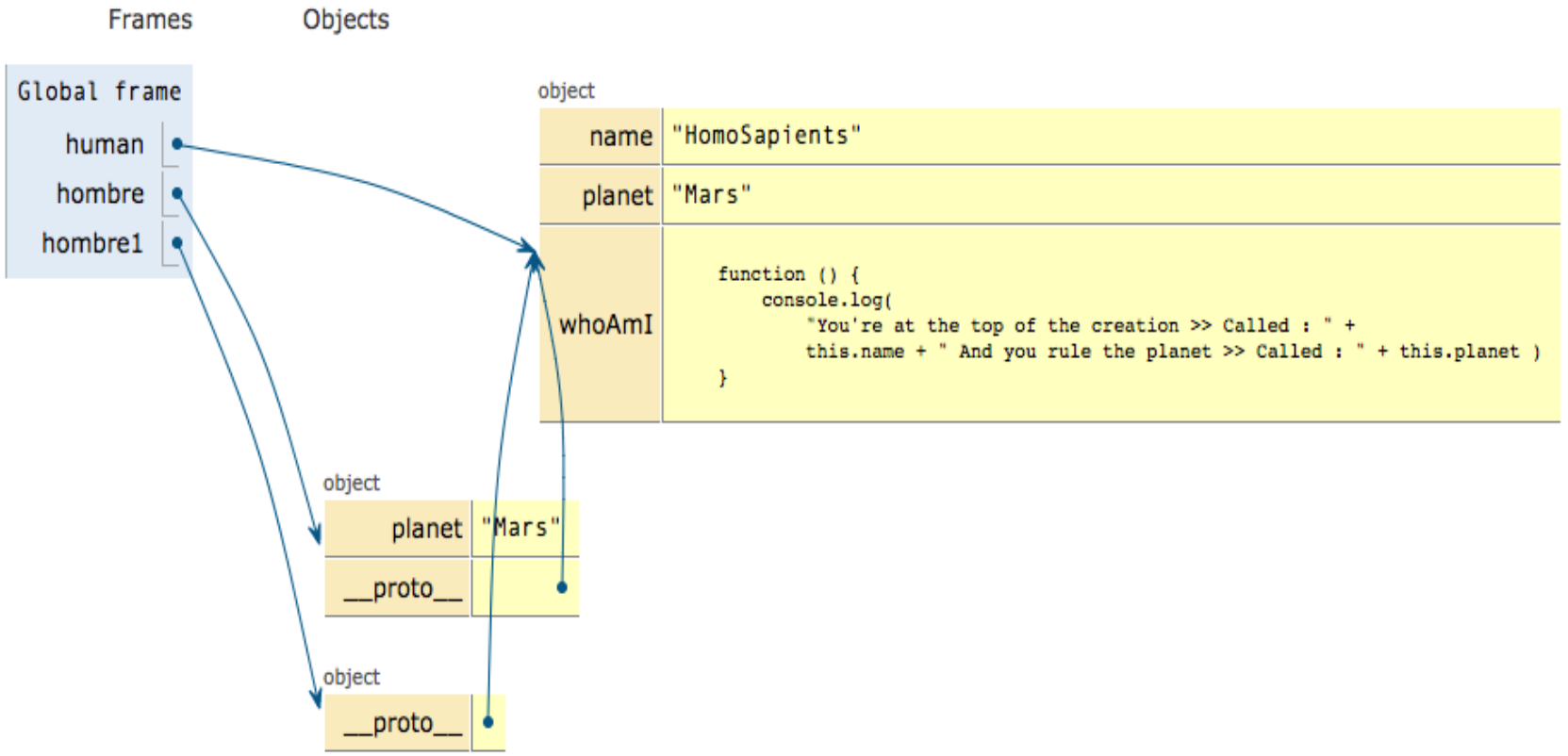
Every Object has a prototype



Object.create method



Visulization of the relationship



THE ARRAY TYPE

- An array is a collection of like values, called elements, such as an array of colors, an array of strings, an array of images. In JavaScript, arrays are built-in objects.
- ECMAScript arrays are ordered lists of data, but unlike in other languages, they can hold any type of data in each slot.
- ECMAScript arrays are also dynamically sized, automatically growing to accommodate any data that is added to them.
- If you know the number of items that will be in the array, you can pass the count into the constructor, and the length property will automatically be created with that value. //var colors = new Array(20);
- The Array constructor can also be passed items that should be included in the array.
- An array can be created with a single value by passing it into the constructor.
- The second way to create an array is by using array literal notation. An array literal is specified by using square brackets and placing a comma-separated list of items between them.
- To get and set array values, you use square brackets and provide the zero-based numeric index of the value.
- The number of items in an array is stored in the length property, which always returns 0 or more.
- A unique characteristic of length is that it's not read-only. By setting the length property, you can easily remove items from or add items to the end of the array.

LENGTH PROPERTY …

- If the length were set to a number greater than the number of items in the array, the new items would each get filled with the value of undefined.
- The length property can also be helpful in adding items to the end of an array.
- The new length is automatically calculated when an item is placed into a position that's outside of the current array size, which is done by adding 1 to the position

Conversion Methods

- All objects have toLocaleString(), toString(), and valueOf() methods.
- The toString() and valueOf() methods return the same value when called on an array.
- It's possible to construct a string with a different separator using the join() method. The join() method accepts one argument, which is the string separator to use, and returns a string containing all items

Stack Methods

- An array object can act just like a stack, which is one of a group of data structures that restrict the insertion and removal of items.
- A stack is referred to as a last-in-first-out (LIFO) structure, meaning that the most recently added item is the first one removed.
- The insertion (called a push) and removal (called a pop) of items in a stack occur at only one point: the top of the stack. ECMAScript arrays provide push() and pop() specifically to allow stack-like behavior.
- The push() method accepts any number of arguments and adds them to the end of the array, returning the array's new length.
- The pop() method, on the other hand, removes the last item in the array, decrements the array's length, and returns that item.

Queue Methods

- A queue adds items to the end of a list and retrieves items from the front of the list. Because the push() method adds items to the end of an array, all that is needed to emulate a queue is a method to retrieve the fIrst item in the array.
- The array method for this is called shift(),which removes the first item in the array and returns it, decrementing the length of the array by one.
- Using shift() in combination with push() allows arrays to be used as queues

Reordering Methods

- Two methods deal directly with the reordering of items already in the array: reverse() and sort().
- The reverse() method simply reverses the order of items in an array.
- By default, the sort() method puts the items in ascending order — with the smallest value first and the largest value last. To do this, the sort() method calls the String() casting function on every item and then compares the strings to determine the correct order.

Manipulation Methods

- Three prominent methods: concat(),slice().splice().
- concat() method allows you to create a new array based on all of the items in the current array.
- This method begins by creating a copy of the array and then appending the method arguments to the end and returning the newly constructed array.
- slice(), creates an array that contains one or more items already contained in an array. The slice() method may accept one or two arguments: the starting and stopping positions of the items to return. If only one argument is present, the method returns all items between that position and the end of the array. If there are two arguments, the method returns all items between the start position and the end position, not including the item in the end position.

Manipulation Methods Splice()

- Perhaps the most powerful array method is splice(), which can be used in a variety of ways. The main purpose of splice() is to insert items into the middle of an array, but there are three distinct ways of using this method.
- Deletion — Any number of items can be deleted from the array by specifying just two arguments: the position of the fi rst item to delete and the number of items to delete. For example, splice(0, 2) deletes the first two items.
- Insertion — Items can be inserted into a specific position by providing three or more arguments: the starting position, 0 (the number of items to delete), and the item to insert. Optionally, you can specify a fourth parameter, fi fth parameter, or any number of other parameters to insert. For example, splice(2, 0, “red”, “green”) inserts the strings “red” and “green” into the array at position 2.
- Replacement — Items can be inserted into a specific position while simultaneously deleting items, if you specify three arguments: the starting position, the number of items to delete, and any number of items to insert.

Location Methods

- ECMAScript 5 adds two item location methods to array instances: indexOf() and lastIndexOf().
- Each of these methods accepts two arguments: the item to look for and an optional index from which to start looking.
- The indexOf() method starts searching from the front of the array (item 0) and continues to the back.
- lastIndexOf() starts from the last item in the array and continues to the front.
- The methods each return the position of the item in the array or –1 if the item isn't in the array.
- An identity comparison is used when comparing the first argument to each item in the array, meaning that the items must be strictly equal as if compared using ===.

Iterative Methods

- ECMAScript 5 defines five iterative methods for arrays.
- Each of the methods accepts two arguments: a function to run on each item and an optional scope object in which to run the function (affecting the value of this).
- The function passed into one of these methods will receive three arguments: the array item value, the position of the item in the array, and the array object itself.
- every() — Runs the given function on every item in the array and returns true if the function returns true for every item.
- filter() — Runs the given function on every item in the array and returns an array of all items for which the function returns true.
- forEach() — Runs the given function on every item in the array. This method has no return value.
- map() — Runs the given function on every item in the array and returns the result of each function call in an array.
- some() — Runs the given function on every item in the array and returns true if the function returns true for any one item.

OBJECT CREATION

- **Although using the Object constructor or an object literal are convenient ways to create single objects, there is an obvious downside: creating multiple objects with the same interface requires a lot of code duplication.To solve this problem, developers began using a variation of the factory pattern.**

The Factory Pattern

- **The factory pattern is a well-known design pattern used in software engineering to abstract away the process of creating specific objects. With no way to define classes in ECMAScript, developers created functions to encapsulate the creation of objects with specific interfaces.**

The Constructor Pattern

- There are native constructors, such as Object and Array, which are available automatically in the execution environment at runtime. It is also possible to define custom constructors that define properties and methods for your own type of object.